

Auswahl der Entwicklungsumgebung

Echtzeitsysteme 2 – Vorlesung/Übung

Peter Ulbrich
Fabian Scheler
Wolfgang Schröder-Preikschat

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme
Friedrich-Alexander Universität Erlangen-Nürnberg

<http://www4.cs.fau.de/~{scheler,ulbrich,wosch}>
{ulbrich,scheler,wosch}@cs.fau.de



1

Überblick

- Einleitung
- Ziele
- Entscheidungsfindung
 - Hardware, Mikrocontroller
 - Programmiersprache, Compiler
 - Betriebssystem

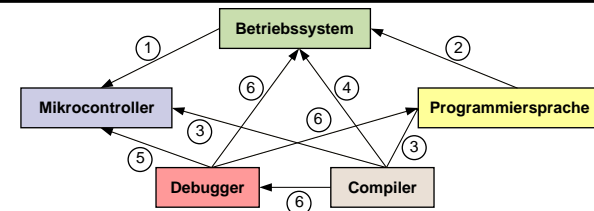
© {ulbrich,scheler,wosch}@cs.fau.de - EZL (SS 2010)

2

Einleitung

- Auswahl einer geeigneten Kombination aus
 - Mikrocontroller
 - Programmiersprache
 - Compiler
 - Betriebssystem
 - Debuggerist **schwierig**
- auch (oder vor allem) ohne **externe Randbedingungen**
 - zwischen den einzelnen Komponenten existieren Abhängigkeiten
- externe Randbedingungen können die Auswahl
 - vereinfachen,
 - erschweren oder
 - unmöglich machen

Interne Abhängigkeiten



A → B
Die Wahl von A beeinflusst
die Wahl von B

- 1) Gibt es ein Betriebssystem für den μ Controller? Welche Eigenheiten des μ Controller werden unterstützt?
- 2) Bietet das Betriebssystem eine Schnittstelle in der passenden Programmiersprache?
- 3) Gibt es einen Compiler der diese Programmiersprache für einen bestimmten μ Controller übersetzt?
- 4) Kann ich das Betriebssystem mit einem diesem Compiler übersetzen?
- 5) Gibt es für den μ Controller passende Debugger?
- 6) Unterstützt der Debugger die Ausgabe des Compilers, das Betriebssystem und die Programmiersprache?

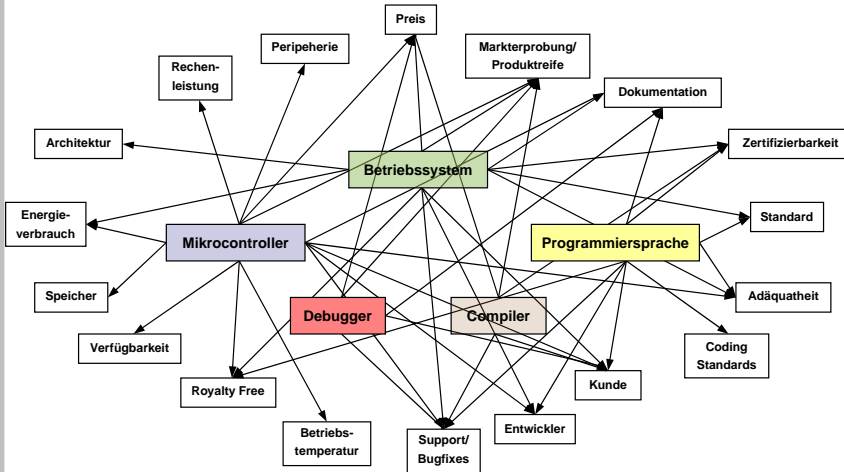
© {ulbrich,scheler,wosch}@cs.fau.de - EZL (SS 2010)

4

© {ulbrich,scheler,wosch}@cs.fau.de - EZL (SS 2010)

3

Externe Abhängigkeiten



© (ulbrich,scheler,wosch)@cs.fau.de - EZL (SS 2010)

5

Auswahlmöglichkeiten - µController



■ dies ist nur ein Bruchteil

© (ulbrich,scheler,wosch)@cs.fau.de - EZL (SS 2010)

6

Auswahlmöglichkeiten - OS



■ dies ist nur ein Bruchteil

© (ulbrich,scheler,wosch)@cs.fau.de - EZL (SS 2010)

7

Ziele ...

- dieses Auswahlprozesses:
 - Optimierung gewisser Metriken
 - Performanz (Durchsatz, mittlere Antwortzeit, ...)
 - Kosten (pro Stück, nonrecurring engineering costs (NRE))
 - Energieverbrauch
 - Time-to-Market
 - Safety
 - ...
- dieser Vorlesung
 - Skizze einer Entscheidungshilfe
 - Aufzeigen einiger relevanter Eigenschaften

© (ulbrich,scheler,wosch)@cs.fau.de - EZL (SS 2010)

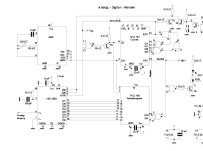
8

Hardware / µController

- **Prozessortechnologie:**
 - **Vielzweckprozessor** vs. **Spezialzweckprozessor**
- **Speicher:**
 - **schreibbar** vs. **nicht schreibbar**
 - **flüchtig** vs. **nicht flüchtig**
- **Speicherarchitektur:**
 - **von Neumann** vs. **Harvard**
- **Instruction Set Architecture (ISA):**
 - **CISC** vs. **RISC**



Prozessortechnologie

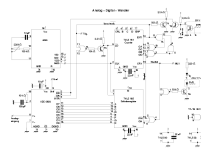


■ **Vielzweck-Prozessor**

- Standardprozessor, z.B. Pentium
- große Teile des Systems entstehen in Software
- + hohe **Verfügbarkeit**, kurze **Time-to-Market**
- + geringe **NRE-Kosten**
- + hohe **Flexibilität**
- ~ Stückkosten: gering – hoch
- ~ Performanz: gering – hoch
- **Energieverbrauch**
- **Chipgröße**



Prozessortechnologie

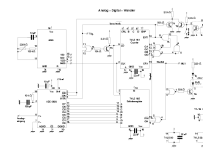
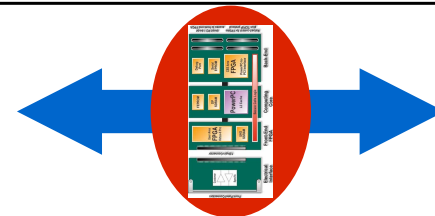


■ **Spezialzweck-Prozessor**

- erfüllt nur eine Aufgabe, z.B. JPEG Codec, Kryptographie
- wenig Software zur Realisierung des Systems erforderlich
- + geringe **Größe**
- + geringer **Energieverbrauch**
- ~ Stückkosten: gering – hoch
- hohe **NRE-Kosten**
- geringe **Flexibilität**
- lange **Entwicklungszeit**



Prozessortechnologie

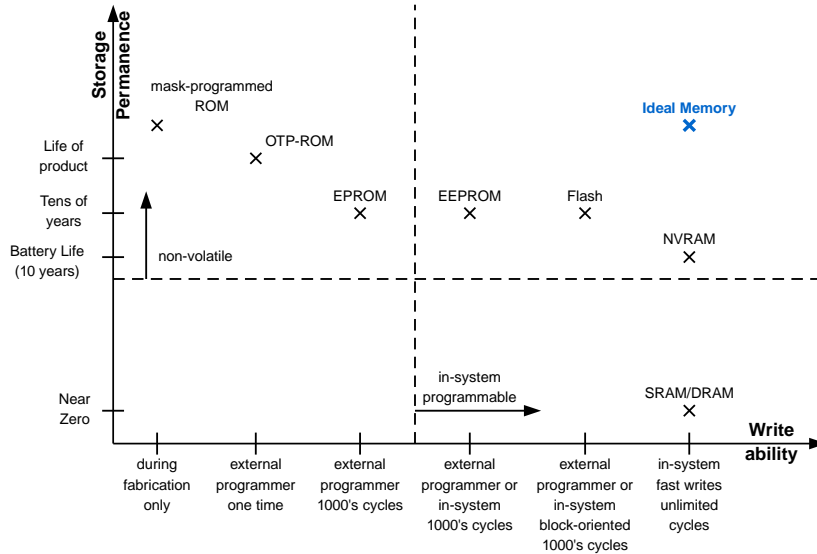


■ **Kompromiss: anwendungsspezifischer Prozessor**

- optimierte Prozessoren (z.B. µController, DSP)
- Kombinationen aus Prozessoren und FPGA / ASIC
- + **Performanz**
- + **Energieverbrauch**
- hohe **NRE-Kosten**
- lange **Entwicklungszeit**
- Unterstützung durch **Entwicklungswerkzeuge**



Speicher



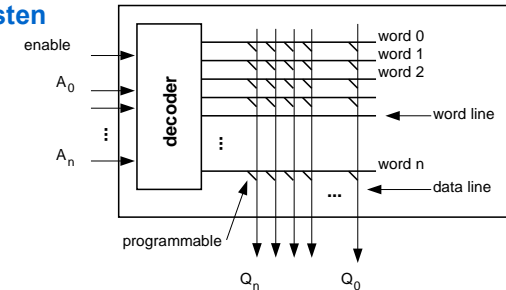
Speichertypen (1)

mask-programmed ROM

- Programmierung durch entsprechende Belichtung des Siliziums
- sehr **unflexibel**, hohe **NRE-Kosten**
- sehr **stabil**

one-time programmable (OTP) ROM

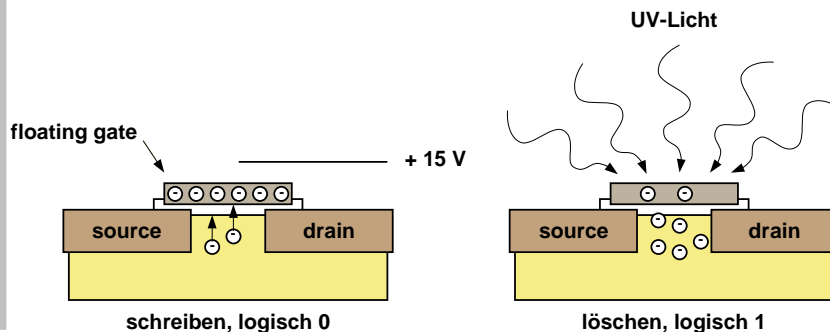
- externes Programmiergerät
- *fusible link* – Programmierung durch Durchtrennen von Sicherungen
- sehr **geringe Stückkosten**
- sehr **stabil**
- **unflexibel**



Speichertypen (2)

erasable programmable ROM (EPROM)

- externes Programmiergerät (*floating gate*)
- löschar durch Bestrahlung mit UV-Licht (ca. 5 – 30 min)
- anfällig für **radioaktive Strahlung** und **elektrische Felder**
- werden selten in der Produktion eingesetzt



Speichertypen (3)

electrically erasable programmable ROM (EEPROM)

- externes Programmiergerät, in-system
- **elektronisch schreib- und löschar** (wortweise)
- **weniger anfällig** für radioaktive Strahlung und elektrische Felder
- ansonsten wie EPROM

Flash memory

- elektronisch schreib- und löschar (blockweise)
- größere Bereiche können **schneller** geschrieben/gelöscht
- ansonsten wie EPROM

Speichertypen (4)

- **static RAM (SRAM)**
 - ein Flip-Flop je Bit: 6 Transistoren
 - kein Refresh notwendig: static RAM
- **dynamic RAM (DRAM)**
 - ein Transistor + ein Kondensator pro Bit: kompakter als SRAM
 - Speicher verliert Inhalt: Refresh notwendig
- **pseudo static RAM (PSRAM)**
 - DRAM mit integriertem Refresh-Controller
- **nonvolatile RAM (NVRAM)**
 - batterie-gestütztes RAM
- Forschung: **MRAM** bzw. **FeRAM**
 - finden langsam ihren Platz in Anwendungen

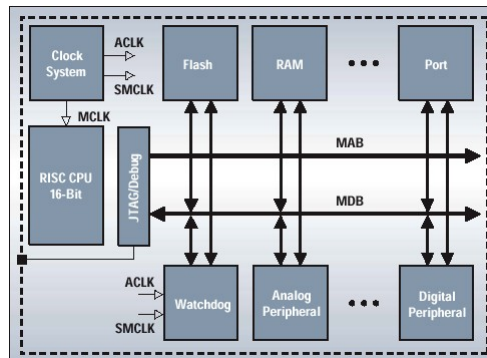
Speicherarchitektur

- zwei verschiedene Anwendungsgebiete
 - Datenverarbeitung
 - mathematische Berechnungen
- **Datenverarbeitung**
 - Beispiele: Textverarbeitung, Betriebssystem, Datenbanken
 - Hauptaufgaben: Daten verschieden, Bedingungen prüfen
 - Charakteristik: tendenziell **sequentiell**
- **mathematische Berechnungen**
 - Beispiele: digitale Signalbearbeitung, Simulation
 - Hauptaufgaben: Addition, Multiplikation
 - Charakteristik: tendenziell **parallel**

von Neumann Architektur

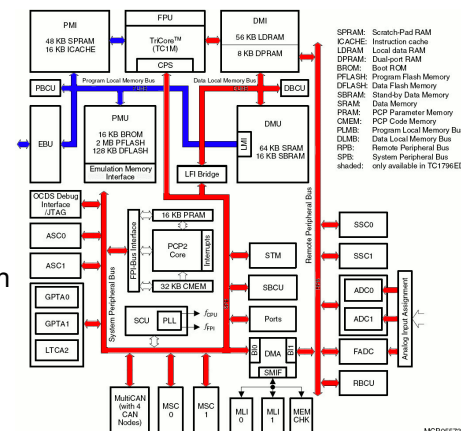
- **gemeinsamer Speicherbereich**
 - Daten und Programm finden sich im selben Speicher
 - **gemeinsamer Bus**
 - Daten und Programm gehen über denselben Bus
- **Flaschenhals!!!**

- Vorteil: billiger
- Beispiel: MSP430



Harvard Architektur

- **getrennte Speicherbereiche**
 - Daten und Programm finden sich in verschiedenen Speicherbereichen
 - teilweise: getrennte Register
 - **getrennte Busse**
 - Daten und Programm gehen über verschiedene Busse
- **weniger Konkurrenz** um den Bus zwischen Daten und Programm
- Beispiel: TriCore

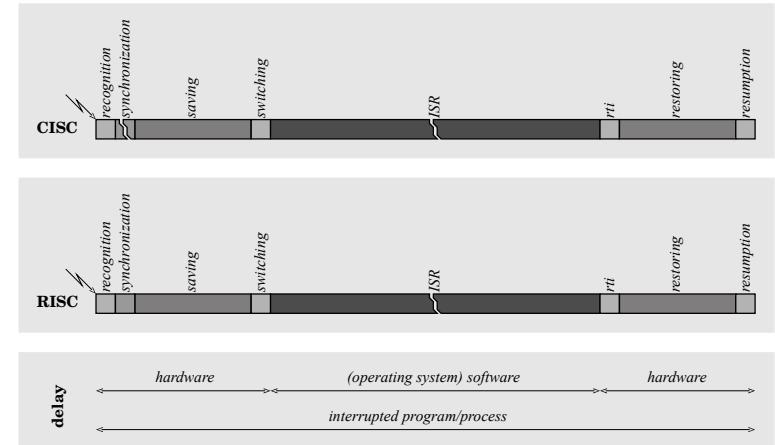


Instruction Set Architecture

- **Complex Instruction Set Computing** (CISC)
 - mächtige Befehle, umfangreicher Befehlssatz
 - zahlreiche Adressierungsarten
 - Konsequenzen
 - + kompakter Programmcode
 - ~ Befehle belegen unterschiedlich viel Speicher
 - Befehle dauern unterschiedlich lange
- **Reduced Instruction Set Computing** (RISC)
 - relativ einfache Befehle, überschaubarer Befehlssatz
 - LOAD / STORE Architektur
 - Konsequenzen
 - + Befehle dauern alle gleich lange
 - ~ Befehle belegen alle gleich viel Speicher
 - größerer Programmcode
- einzelne Befehle sind i.d.R. nicht unterbrechbar



ISA - Interruptlatenz



weitere Entscheidungsgrundlagen

- IC-Technologie:
 - full custom / VLSI vs. CPLD / FPGA
- Kommunikationssysteme:
 - time-triggered vs. event-triggered
- Peripherie:
 - Analog-Digital-Wandler, Digital-Analog-Wandler
 - SPI, I2C, Timer, PWM, CAN, LIN, ...
 - I/O-Pins: 3.3V vs. 5V
- Package:
 - Wie viele Pins hat der Chip letztendlich?
 - SMD – ja oder nein?
 - BGA oder Pins?



Betriebssystem

- Grundsätzliche Frage
 - Common of the shelf (COTS) RTOS, Eigenbau oder keines?
- **Eigenbau**
 - + schont **Ressourcen** (RAM, ROM, ...)
 - + exakt auf das System **zugeschnitten**
 - hoher **Entwicklungsaufwand**, hohe **NRE**
- **COTS RTOS**
 - + geringer **Entwicklungsaufwand**, geringe **NRE**
 - + **Flexibilität**
 - **Overhead**
 - höhere **Stückkosten**



Slicing the operating systems pie

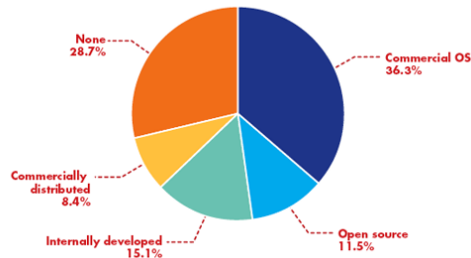


Figure 1

Warum kein OS?

- 85%: „we don't need one“
- 30%: Overhead
- 10%: zu teuer
- 7%: komplizierte Verwendung

Welche Art von OS?

- 51%: kommerziell (2005: 44%)
- ersetzen Eigenentwicklungen

Wer setzt ein OS ein?

- 33%: Automotive
- 50%: Durchschnitt
- 64%: Luftfahrt

■ grundsätzlicher Ansatz (Laplante)

- wähle Kriterien c_1 bis c_n , mit $0 \leq c_i \leq 1$; $1 \leq i \leq n$
- wähle Gewichte w_1 bis w_n , mit $0 \leq w_i \leq 1$; $1 \leq i \leq n$
- 0 – nicht erfüllt bzw. nicht relevant,
1 – komplett erfüllt bzw. sehr relevant
- Bewertung des RTOS: $\sum_{i=1}^n c_i \cdot w_i$

■ **Problematic:** verlässliche Information finden

- Angaben der Hersteller sind oft für Marketingzwecke *frisirt*
- Erfahrungsberichte anderer Anwender: aufwendige Recherche

■ die Wahl des richtigen Betriebssystems ist entscheidend und sollte nicht zu einer *make-or-break* Entscheidung werden!

RTOS: Kriterien (1)

■ minimale Interrupt-Latenz, c_1

Zeit vom Auftreten einer Unterbrechung bis zur Ausführung der ersten Anweisung der benutzerdefinierten Behandlungsroutine.

■ Anzahl der unterstützten Fäden, c_2

Man benötigt eine gewisse Anzahl von Fäden, um alle Ereignisbehandlungen abbilden zu können. Die Anzahl der Fäden wird dabei nicht nur durch das Betriebssystem limitiert, sondern auch durch den zur Verfügung stehenden Speicher.

■ Speicherbedarf des Betriebssystems, c_3

Der Speicherbedarf des Betriebssystems allein, etwaige Anwendungsprogramme spielen keine Rolle!

RTOS: Kriterien (2)

■ zur Auswahl stehende Scheduling-Verfahren, c_4

Verschiedene Scheduling-Verfahren erlauben es, das System besser an die Anwendung anzupassen.

■ zur Auswahl stehende IPC-Verfahren, c_5

Manche IPC-Mechanismen sind zur Formulierung gewisser Probleme einfach besser geeignet als andere, es erleichtert dem Benutzer die Arbeit, wenn er einen passenden Mechanismus auswählen kann.

■ Support, c_6

Das beste Betriebssystem ist nutzlos, wenn man es nicht benutzen kann - Hotline, Schulungen, Mailinglisten, Dokumentation ...

RTOS: Kriterien (3)

■ zur Verfügung stehende Anwendungen, c_7

Existieren Anwendungen oder Bibliotheken, die ich in meinem System wieder-verwenden kann? Dies ist besonders im Echtzeitumfeld schwierig.

■ unterstützte Architekturen bzw. Peripherie, c_8

Gibt es Portierungen dieses Betriebssystems für verschiedene Prozessor-typen, welche Peripheriegeräte werden standardmäßig unterstützt? Wie viel Handarbeit ist notwendig, damit das Betriebssystem die Anwendung vollständig unter-stützt?

■ Ist das Betriebssystem quelloffen?, c_9

Ist der Quelltext des Betriebssystems für den Entwickler les- oder sogar verän-derbar, um Vorgänge im Betriebssystem besser nachvollziehen oder anpassen zu können? Wird das Betriebssystem als Binärbild ausgeliefert?

RTOS: Kriterien (4)

■ Kontextwechsel, c_{10}

Wie viel Zeit benötigt das Betriebssystem um den Wechsel des laufenden Fa-dens durchzuführen?

■ durch das Betriebssystem verursachte Kosten, c_{11}

Zu diesen Kosten zählen sowohl Ausgaben, um Arbeitsplätze mit den entspre-chen den Entwicklungswerkzeugen auszustatten, als auch später in der Produk-tion des Systems anfallende Lizenzkosten.

■ zur Verfügung stehenden Entwicklungswerkzeuge, c_{12}

Gibt es Werkzeuge, die den Entwickler beim Umgang mit dem Betriebssystem unterstützen? Implementiert das Betriebssystem einen Standard und ist es ge-gen anderen Implementierungen austauschbar?

RTOS: ProOSEK/time vs. eCos

Kriterium				
Nr.	Beschreibung	Gewichtung	ProOSEK	eCos
1	Interruptlatenz	1	1	0,7
2	Anzahl der Fäden	1	1	1
3	Speicherbedarf	1	1	0,6
4	Scheduling	0,8	1	1
5	IPC	1	1	1
6	Support	1	0,6	1
7	Anwendungen	0,5	0	1
8	Portabilität	0,1	0,7	1
9	Quellcode	0,1	0	1
10	Kontextwechsel	0,5	1	0,75
11	Kosten	0,1	0	1
12	Standards	0,1	1	1
Ergebnis			5,97	6,19

Embedded.com: OS Evaluation [1]

Operating systems evaluation criteria

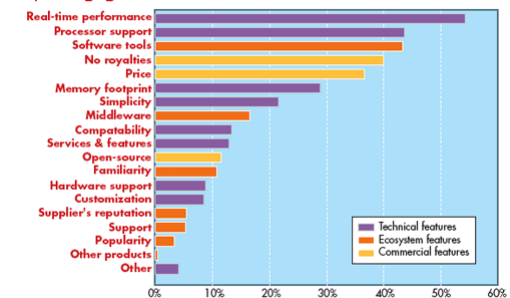


Figure 5

■ Processor Support

- pass/fail Kriterium

■ Performanz vor Kompatibilität

■ Werkzeuge werden wichtiger

- keine Beziehung zum OS

Embedded.com: Open Source [1]

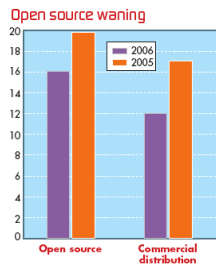


Figure 2

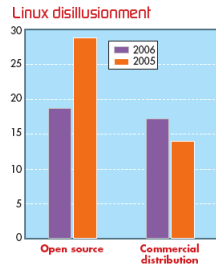


Figure 3

- Wer verwendet Linux?
 - 38%: Automatisierung
- Wer verwendet Linux nicht?
 - 88%: Automotive & Luftfahrt
- Pro Linux
 - 70%: geringer Preis
 - 61%: Anpass-/Erweiterbarkeit
 - 45%: Kontrolle der Features
- Contra Linux
 - 60%: Portierungsaufwand
 - 31%: Real-Time Performance
 - 28%: Support

Programmiersprache

„Misuse of the underlying programming language can be the single greatest source of performance deterioration and missed deadlines in real-time systems.“

Phillip A. Laplante

Eignung einer Programmiersprache

- Metriken (informell) nach Cardelli:
 - **Economy of Execution**
Wie schnell wird ein Programm ausgeführt? Ist die Programmiersprache inhärent mit einem Performanznachteil verbunden?
 - **Economy of Compilation**
Wie schnell kann man ein Programm übersetzen?
 - **Economy of Small-Scale Development**
Wie schwierig ist der Umgang mit der Programmiersprache für einen einzelnen Programmierer?
 - **Economy of Large-Scale Development**
Wie schwierig ist der Umgang mit der Programmiersprache für Team von Programmierern?
 - **Economy of Language Features**
Wie schwer ist es, die Programmiersprache zu erlernen?
- zusätzlich in Echtzeitsystemen: **Analysierbarkeit**

Assemblersprachen

- Beispiele: je nach Prozessor
 - + direkte Kontrolle über die Hardware
 - kaum Abstraktionsmechanismen
 - schwierig, fehlerbehaftet, aufwendig
 - unportabel
- ➔ Assembler nur im Ausnahmefall verwenden!

Prozedurale Sprachen

- Beispiele: C, Fortran, Ada95, Modula-2
 - + Strukturierung der Programme in Unterprogramme
 - + Typisierung / abstrakte Datentypen
 - + Ausnahmebehandlung
 - + dynamische Speicherverwaltung
 - ~ unterschiedlich stark ausgeprägte Modulkonzepte
 - Kapselung
- in Echtzeitsystemen am häufigsten anzutreffen

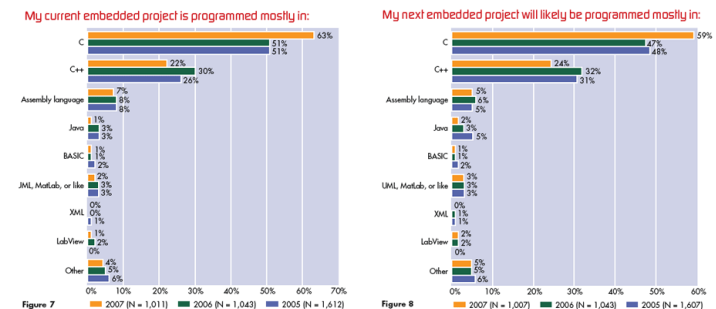
Objektorientierte Sprachen

- Beispiele: C++, C#, Java, Ada95
 - statisch übersetzt vs. dynamisch ausgeführt
 - + Kapselung und Information Hiding
 - ~ Vererbung und Polymorphismus
 - ~ Ausnahmebehandlung
 - ~ Garbage Collection
- in Echtzeitsystemen (noch) selten eingesetzt, auch, weil fähige Entwickler (noch) fehlen

Metriken

		prozedural			objekt-orientiert		
	Assembler	C	Fortran	Ada95	C++	Ada95	Java
Execution	++	++	++	++	+	o	o
Compilation	++	++	++	++	+	+	+
Small-Scale Development	-	++	+	++	+	+	+
Large-Scale Development	-	o	-	+	++	++	++
Language Features	++	+	++	+	-	-	-
Analysierbarkeit	++	-	++	+	-	o	-

Embedded.com: Languages [2]



- Grob gesagt: C gewinnt auf Kosten von C++ → warum?
 - Outsourcing: C++ stellt höhere Ansprüche an Entwickler
 - C++ könnte bei der Wiederverwendbarkeit helfen
 - aber große Teil des Codes hängen an der Hardware
 - Alles in Allem kommt man hier mit C gut zurecht
 - „C++ tends to introduce a lot of variables and make the project more complex“

Zusammenfassung

- Auswahl der geeigneten Entwicklungsumgebung ist schwierig
 - es existieren eine Fülle interner und externer Abhängigkeiten
- **Hardware**
 - Vielzweck-Prozessor vs. Spezialzweck-Prozessor
 - diverse Speichertypen
 - von Neumann vs. Harvard
 - RISC vs. CISC
- **Betriebssystem**
 - Eigenbau vs. COTS
 - schwierige Informationsbeschaffung
 - Metrik: Gewichtung einer Menge von Kriterien
- **Programmiersprachen**
 - Assembler, prozedurale Sprachen, objekt-orientierte Sprachen
 - Metriken nach Cardelli



Literatur

1. *Jim Turley.*
Operating systems on the rise.
Embedded Systems Design, 2006.
2. *Richard Nass.*
Annual study uncovers the embedded market.
Embedded Systems Design, 2007.

